Problem
Given an $n \times 2$ pavement, how many ways can you tile it using dominoes?
Let us look at the last two tiles, $A, B$, of the pavement

\[
\begin{array}{|c|c|}
\hline
\text{A} & \text{B} \\
\hline
\end{array}
\]

These can be covered using either one vertically placed tile or two horizontally placed tiles.

In the first case, we are left to cover a pavement of size $(n - 1) \times 2$ and in the second case, we are left with $(n - 2) \times 2$. 
Let $f(n)$ be the number of ways to cover this pavement with dominoes. We have that

$$f(n) = f(n - 1) + f(n - 2)$$

and we also see that $f(1) = 1, f(2) = 2$. Which means that $f(n) = F_n$, the n’th Fibonacci number. So given $n$, how long does it take us to calculate $F_n$?
Algorithm 1 Recursive Fibonacci

function recursive_F(Int n)
    if n=0 or n=1 then
        return 1
    elseif
        then return recursive_F(n-1) + recursive_F(n-2)
    end if
end function
Time complexity of recursion: \( T_r(n) \)

\[
T_r(n) = T_r(n - 1) + T_r(n - 2) \\
\geq 2 \cdot T_r(n - 2) \\
\geq 4 \cdot T_r(n - 4) \\
\vdots \\
\geq 2^k T_r(n - 2k) \quad \text{Induction hypothesis} \\
\vdots \\
\geq 2^{n/2} T(0) = 2^{n/2}
\]

Using naive recursion, we can’t calculate the 100’th Fibonacci number, which takes \( 2^{50} \approx 10^{15} \) steps. Can we do better?
Let us use a cleverer looping strategy, using “memoization”.

Algorithm 2 Memoized Fibonacci

function memoized_f(Int n)
    if n=0 or n=1 then
        return 1
    else
        \( f_0 \leftarrow 1, f_1 \leftarrow 1, f_2 \leftarrow 2, s \leftarrow n - 1 \)
        while \( s > 0 \)
            \( f_0 \leftarrow f_1, f_1 \leftarrow f_2 \)
            \( f_2 \leftarrow f_0 + f_1 \)
            \( s \leftarrow s - 1 \)
        end while
        return \( f_2 \)
    end if
end function
Time complexity of memoization: $T_m(n)$ is $O(n)$, as we only have one while loop, which goes from $n - 1 \to 0$. 
Time complexity of memoization: $T_m(n)$ is $O(n)$, as we only have one while loop, which goes from $n - 1 \to 0$.

What is the time complexity in terms of the input size?
Time complexity of memoization: $T_m(n)$ is $O(n)$, as we only have one while loop, which goes from $n - 1 \rightarrow 0$.
What is the time complexity in terms of the input size?
Defining input size: We get a single number $n$ as input, which has $\log_2 n$ bits in its binary representation.
So the input size is $O(\log n)$. 
Time complexity of memoization: \( T_m(n) \) is \( O(n) \), as we only have one while loop, which goes from \( n - 1 \rightarrow 0 \).

What is the time complexity in terms of the input size?

Defining input size: We get a single number \( n \) as input, which has \( \log_2 n \) bits in its binary representation.

So the input size is \( O(\log n) \).

The time complexity for the memoized solution is \( O(2^{\log n}) \), which is exponential in the input size.

Can we do better?
Let us look at the following matrix, $\mathcal{F}$, and vector, $I$,

\[
\mathcal{F} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad I = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]

What happens when we multiply $\mathcal{F} \cdot I$

\[
\mathcal{F} \cdot I = \begin{bmatrix} 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\]

on further multiplication

\[
\mathcal{F}^2 \cdot I = \begin{bmatrix} 1 \cdot 1 + 1 \cdot 1 \\ 1 \cdot 1 + 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}
\]
Let us multiply some more

\[ \mathcal{F}^3 \cdot I = \begin{bmatrix} 1 \cdot 2 + 1 \cdot 1 \\ 1 \cdot 2 + 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \]

Claim

\[ \mathcal{F}^n \cdot I = \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \quad \forall \ n \]

Proof by induction

\[ \mathcal{F}^{n+1} \cdot I = \mathcal{F} \cdot \mathcal{F}^n \cdot I \]

\[ = \mathcal{F} \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \quad \text{induction hypothesis} \]

\[ = \begin{bmatrix} 1 \cdot F_n + 1 \cdot F_{n-1} \\ 1 \cdot F_n + 0 \cdot F_{n-1} \end{bmatrix} \]

\[ = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} \]
How does that help us?
How does that help us?
Claim: We can calculate \( a^n \) in \( O(\log n) \). E.g.

\[
3^{32} = (3^2)^{16} = ((3^2)^2)^8 = (((3^2)^2)^2)^2
\]

Squaring is just one multiplication, so we can go from
3 \( \rightarrow \) 9 \( \rightarrow \) 81 \( \rightarrow \) 6561 \( \rightarrow \) \( \cdots \) \( \rightarrow \) 1853020188851841 in 5 steps instead of 32.
How to do for general $n$ which is not a power of 2?

Algorithm 3 Repeated Squaring

```plaintext
function power(Int a, Int n)
    if n=0 then
        return 1
    else
        r ← power(a, ⌊n/2⌋)
        r ← r · r
        if n is odd then
            r ← r · a
        end if
        return r
    end if
end function
```
Now, we can do the same for the matrix $F$, to get $F^n$ and calculate the $n$’th Fibonacci number in $O(\log n)$.
Now, we can do the same for the matrix $\mathcal{F}$, to get $\mathcal{F}^n$ and calculate the $n$’th Fibonacci number in $O(\log n)$.

Algorithm 5 Matrix Fibonacci

```plaintext
function matrix_F(Int n)
    $\mathcal{F} \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
    $S \leftarrow \text{power}(\mathcal{F}, n)$
    $\mathcal{V} \leftarrow S \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
    return $\mathcal{V}[0]$
end function
```
But there is a catch.
How large is the $n$’th Fibonacci number?
But there is a catch.
How large is the $n'$th Fibonacci number?
We have that

\[ F_n = \frac{\Phi^n - \phi^n}{\sqrt{5}} \]

\[ \approx \frac{\Phi^n}{\sqrt{5}} \]

\[ = \frac{(\Phi^2)^{n/2}}{\sqrt{5}} \]

\[ \geq \frac{2^{n/2}}{\sqrt{5}} \]

Which implies that the $n'$th Fibonacci number has $O(n)$ bits in its binary representation.
Hence we cannot do better than $O(n)$, even if we are only doing $O(\log n)$ multiplications.
Workaround: Instead of finding the full Fibonacci number, find the last $k$ digits instead.

E.g for $k = 6$

$$F_{100} = 354224848179261915075 \rightarrow 915705$$

Now the output is bounded in size, assuming that $k$ is predefined constant.

But if we calculate the Fibonacci numbers in the intermediate steps, we would still be using $O(n)$ memory.

We need to modify the whole algorithm to only keep track of the last $k$ digits.
Workaround: Instead of finding the full Fibonacci number, find the last $k$ digits instead.
E.g for $k = 6$

$$F_{100} = 354224848179261915075 \rightarrow 915705$$

Now the output is bounded in size, assuming that $k$ is predefined constant.
But if we calculate the Fibonacci numbers in the intermediate steps, we would still be using $O(n)$ memory.
We need to modify the whole algorithm to only keep track of the last $k$ digits.
How do we do all calculations while keeping only $k$ digits?
Modular Arithmetic
Definition
We call two integers $x$ and $y$ “congruent” modulo an integer $n$ if $n$ divides $x - y$.
We write it as

$$x \equiv y \pmod{n}$$

E.g.

$$2 \equiv 10 \pmod{4}$$
$$1 \equiv 100 \equiv 400 \pmod{3}$$
Exercise: Prove that the congruence relation is an equivalence relation. I.e. prove that it is symmetric, reflexive and transitive. The congruence classes of this relation are also called the “residue” classes modulo $n$. 
Addition and subtraction in modular arithmetic.

Problem
Find the last two digits of $12345 + 56789$

Here we have to do calculation modulo 100

$$12345 + 56789 \equiv 45 + 89 \pmod{100}$$

$$\equiv 134 \pmod{100}$$

$$\equiv 34 \pmod{100}$$
Problem

Find the remainder of $97531 - 24680$ modulo 6

$$97531 - 24680 \equiv 1 - 2 \pmod{6}$$

$$\equiv -1 \pmod{6}$$

$$\equiv 5 \pmod{6}$$

We always want the final answer to be positive.
Problem
Find the remainder of $97531 - 24680$ modulo $6$

$$97531 - 24680 \equiv 1 - 2 \pmod{6}$$
$$\equiv -1 \pmod{6}$$
$$\equiv 5 \pmod{6}$$

We always want the final answer to be positive. In general,

$$a \equiv b \pmod{n}$$
$$c \equiv d \pmod{n}$$

$$a \pm c \equiv b \pm d \pmod{n}$$
Find the remainder of $234 \cdot 345$ modulo 8.

$$234 \cdot 345 \equiv 2 \cdot 1 \pmod{8}$$

$$\equiv 2 \pmod{8}$$
Find the remainder of $234 \cdot 345$ modulo 8.

$$234 \cdot 345 \equiv 2 \cdot 1 \pmod{8}$$

$$\equiv 2 \pmod{8}$$

Exercise: Find the remainder of $3^{78}$ modulo 4.
Algorithm 6 Repeated modular squaring

function power(Int a, Int n, Int m)
    # calculates $a^n \pmod{m}$
    if n=0 then
        return 1
    else
        $r \leftarrow power(a, \lfloor n/2 \rfloor, m)$
        $r \leftarrow (r \cdot r) \pmod{m}$
        if n is odd then
            $r \leftarrow (r \cdot a) \pmod{m}$
        end if
        return $r$
    end if
end function

Python implements this in the pow function
Modular division
Let $[0], [1], [2], [3], \ldots, [n - 1]$ be the residue classes of $n$. The set of residue classes $\{[0], [1], \ldots, [n - 1]\}$ is called $\mathbb{Z}_n$. We have already seen how to add, subtract, multiple in $\mathbb{Z}_n$. 
Division is defined as the opposite of multiplication:

\[ \frac{a}{b} := x \quad \exists \quad b \cdot x \equiv a \pmod{n} \]

E.g.

\[
\frac{[2]}{[4]} \equiv [2] \pmod{6} \\
\frac{[4]}{[2]} \equiv [2] \pmod{6} \\
\frac{[2]}{[4]} \equiv [3] \pmod{5}
\]

We will omit the \([\cdot]\) and only refer to the integers, when context is obvious.
Sometimes the division is not possible:

\[ \frac{3}{4} \pmod{6} \]

Can we know apriori when division is going to succeed?
Given $a, b, n$ can we find an $x$ such that $a \equiv b \cdot x \pmod{n}$.

Claim: Let $g = \gcd(b, n)$. If $g | a$ then there exists an $x$ satisfying our conditions.

Let us start in a simple case, where $g = 1$ and we want $x$ such that $b \cdot x \equiv 1 \pmod{n}$.
Assume that there is no $x$ such that $b \cdot x \equiv 1 \pmod{n}$.

Take the set $\{b \cdot 1, b \cdot 2, \ldots, b \cdot (n - 1)\} \subseteq \mathbb{Z}_n \setminus \{1\}$.

Now there are two possibilities for the set $\{b \cdot 1, \ldots, b \cdot (n - 1)\}$.

Either 0 is in this set or it is not.

We will show a contradiction in both cases.
Case 1: $0 \in \{b \cdot 1, \ldots, b \cdot (n - 1)\}$
Which means

\[\exists y \ni b \cdot y \equiv 0 \pmod{n}\]

\[\implies by = nq\quad\text{for some integer } q\]

\[\implies n|by\]

\[\implies n|y\quad\text{as } \gcd(b, n) = 1\]

But this is a contradiction as $y < n$, as it is a residue class of $n$. 
Case 2: $0 \notin \{b \cdot 1, \ldots, b \cdot (n - 1)\}$
Which means that

$$\{b \cdot 1, \ldots, b \cdot (n - 1)\} \subseteq \{2, 3, \ldots, n - 1\}$$

The LHS has $n - 1$ terms and the RHS has $n - 2$ terms.

By pigeon hole principle

$$\exists y_1, y_2 \ni b \cdot y_1 \equiv b \cdot y_2 \pmod{n}$$

$$\implies b(y_1 - y_2) \equiv 0 \pmod{n}$$

$$\implies b \cdot y \equiv 0 \pmod{n}$$

Which is again a contradiction, by the previous argument.
In either case, we have shown that there will be an \( x \) such that 
\[ b \cdot x \equiv 1 \pmod{n}. \]
This element is called the inverse of \( b \), represented by \( b^{-1} \).
To calculate \( a/b \), first calculate \( b^{-1} \) and then calculate \( a \cdot b^{-1} \).

E.g. \( 3^{-1} \equiv 5 \pmod{7} \)
\[ \implies \frac{4}{3} \equiv 4 \cdot 3^{-1} \equiv 4 \cdot 5 \equiv 20 \equiv 6 (\pmod{7}) \]
Which can be verified as \( 3 \cdot 6 \equiv 4 \pmod{7} \).
In either case, we have shown that there will be an $x$ such that $b \cdot x \equiv 1 \pmod{n}$.

This element is called the inverse of $b$, represented by $b^{-1}$. To calculate $a/b$, first calculate $b^{-1}$ and then calculate $a \cdot b^{-1}$.

E.g. $3^{-1} \equiv 5 \pmod{7}$

$\implies \frac{4}{3} \equiv 4 \cdot 3^{-1} \equiv 4 \cdot 5 \equiv 20 \equiv 6 \pmod{7}$

Which can be verified as $3 \cdot 6 \equiv 4 \pmod{7}$.

How do we actually find this inverse?
Euclid's algorithm
Algorithm 7 Euclid's algorithm

function gcd(Int a, Int b)
    if b=0 then
        return a
    else
        return gcd(b, a\%b)
    end if
end function
Proof of correctness:
Follows from the fact that
\[ \gcd(a, b) = \gcd(a - b, b) = \gcd(a - 2b, b) = \ldots = \gcd(a - qb, b) \]
And when we take \( q \) to be \( \lfloor \frac{a}{b} \rfloor \), we get the standard algorithm as above.
How fast is this algorithm?
Proof of correctness:
Follows from the fact that
\[ \gcd(a, b) = \gcd(a - b, b) = \gcd(a - 2b, b) = \ldots = \gcd(a - qb, b) \]
And when we take \( q \) to be \( \lfloor \frac{a}{b} \rfloor \), we get the standard algorithm as above.

How fast is this algorithm?
Let the inputs to the recursive calls be as follows

\[(a, b) = (a_h, b_h) \rightarrow (a_{h-1}, b_{h-1}) \rightarrow \cdots \rightarrow (a_0, b_0) = (g, 0)\]

Which means that from \((a_h, b_h)\) the recursion depth is \( h \).
Claim:

\[ b_i \geq F_i, \quad F_i = i^{th} \text{ Fibonacci number} \]
Claim:

\[ b_i \geq F_i, \quad F_i = i'\text{th Fibonacci number} \]

We prove the claim by induction on the recursion depth, \( h \).

Base case: \( h = 0 \)

We don’t need to make any recursive calls, which means that \( b = 0 \).

\( 0 = b = F_0 \), so our claim is true for the base case.

Induction hypothesis:

If we make \( h \) recursive calls from input \((a_h, b_h)\) then \( b_h \geq F_h \).
Now suppose that we have input \((a_{h+1}, b_{h+1})\), i.e. from this input Euclid’s algorithm made \(h + 1\) recursive calls.

The recursive calls made are

\[(a_{h+1}, b_{h+1}) \rightarrow (a_{h}, b_{h}) \rightarrow (a_{h-1}, b_{h-1}) \rightarrow \cdots\]

By way of recursion, we have

\[
a_h = b_h \cdot q + b_{h-1}
\]

\[
b_{h+1} = b_h \cdot q + b_{h-1}
\]

for some quotient \(q\)

\[b_{h+1} \geq b_h + b_{h-1}\]

as the recursion swaps \(a\) to \(b\)

\[b_{h+1} \geq F_h + F_{h-1}\]

induction hypothesis

\[b_{h+1} \geq F_{h+1}\]

Corollary: Two consecutive Fibonacci numbers are co-prime.
Hence, Euclid’s algorithm runs in $O(\log b)$. Here the size of $a$ doesn’t matter as it gets killed in the first step.
We use Euclid’s algorithm by extending it to find the inverse of $b$
number modulo $n$.
In particular, we note the following Bezout’s identity

$$\forall a, b \; \exists \; s, t \; \exists \; a \cdot s + b \cdot t = \gcd(a, b) = g$$

If $\gcd(a, b) = 1$ then $t = b^{-1}$.
We will extended Euclids algorithm to calculate $s, t$ in addition to $g$. 
Algorithm 8 Extended Euclids Algorithm

function EEuclids(Int a, Int b)
    # returns g, s, t
    if b=0 then
        return (a, 1, 0)
    else
        q ← ⌊a/b⌋, r ← a % b
        g, s', t' ← EEuclids(b, r)
        s ← t
        t ← s' − qt'
        return (g, s, t)
    end if
end function
Proof of correctness:
Base case: \( b = 0 \), in which case the \( \text{gcd} \) is \( g = a \) and \( (s, t) = (10) \)

\[
a \cdot 1 + 0 \cdot 0 = a
\]

Now suppose that the algorithm is true for the recursive call. I.e.

\[
b \cdot s' + r \cdot t' = g
\]

Now let us find the value of \( a \cdot s + b \cdot t \)

\[
a \cdot s + b \cdot t = a \cdot t' + b \cdot (s' - qt')
\]

\[
= bs' - (a - bq)t'
\]

\[
= bs' - rt'
\]

\[
= g \quad \text{induction hypothesis}
\]
But what about the case when \( \gcd(b, n) \neq 1 \)?

E.g.

\[
\frac{50}{30} \equiv 5 \pmod{100}
\]

Or is it 15? or 25?

How many solutions to

\[
50 \equiv 30 \cdot x \pmod{100}
\]

In general to any equation of the form

\[
a \equiv b \cdot x \pmod{n}
\]
In cases where $\gcd(b, n) \neq 1$, we have more than one solution. First, reduce the equation, by dividing through out by $g = \gcd(b, n)$ to get

$$\frac{a}{g} \equiv \frac{b}{g} \cdot x \pmod{\frac{n}{g}}$$

There exists a unique solution to this equation, say $\alpha$. Now all the solution to the original equation are given by

$$\alpha, \alpha + \frac{n}{g}, \alpha + 2\frac{n}{g}, \ldots, \alpha + (g - 1)\frac{n}{g}$$

which in our example, gives us $g = 10, \alpha = 5$ and solutions being

$$5, 15, 25, \ldots, 95$$